Detecting Kernel-Level Rootkits Through Binary Analysis

Christopher Kruegel
<chris@auto.tuwien.ac.at>
Automation Systems Group, Technical University Vienna

William Robertson and Giovanni Vigna
<{wkr,vigna}@cs.ucsb.edu>
Reliable Software Group, UC Santa Barbara

December 8, 2004

Overview

- **Motivation**
- Kernel-Level Rootkit Detection
- System Evaluation
- Conclusions and Future Work

What Are Rootkits?

- Tools used by attackers after compromising a system
  - hide presence of attacker
  - allow for return of attacker at later date
  - gather information about environment
  - attack scripts for further compromises
- Traditionally trojaned set of userland applications
  - system logging (syslogd)
  - system monitoring (ps, top)
  - user authentication (login, sshd)
  - etc.

Kernel-Level Rootkits

- New type of rootkit that modifies system kernel
- Modifies kernel data structures
  - process listing
  - module listing
- Intercepts requests from userspace applications
  - system call boundary
  - VFS fileops struct

Why Are Kernel-Level Rootkits Bad?

- Traditional rootkits easily detected with filesystem integrity checkers
  - e.g., Tripwire
  - kernel, however, controls view of system for userspace applications
- Malicious kernel code can intercept attempts by userspace detector to find rootkits
  - remove rootkit module from listing
  - prevent or modify reads to /dev/kmem
  - etc.
- Thus, theoretically kernel-level rootkits are in the worst case undetectable from userspace

Current Detection Methods

- chkrootkit
  - userspace, signature-based detector
- kstat, rkstat, St. Michael
  - kernelspace, signature-based detector
  - implemented as kernel modules or use /dev/kmem
- Limitations of current detection methods
  - rootkit must be loaded in order to detect it
  - thus, detectors can be thwarted by kernel-level rootkit
  - also suffer from limitations of signature-based detection

Overview

- Motivation
- **Kernel-Level Rootkit Detection**
- System Evaluation
- Conclusions and Future Work

Our Detection Method

- Linux kernel exports well-defined interface to modules
  - observation: kernel rootkits (generally) violate interface
- From defined interface, we extract a specification of allowed modifications of kernel memory
- Statically analyze kernel module binaries to determine whether kernel-module interface is violated
  - i.e., whether module performs writes to invalid kernel addresses
  - analysis performed after module load but before initialization, thus code deemed malicious is never allowed to execute

Behavioral Specifications

- Specifications composed of set of allowed operations legitimate kernel modules may perform
- Examples of legitimate operations
  - registering device with kernel
  - accesses to devices mapped into kernel memory
  - overwriting exported function pointers for event callbacks
- Examples of illegal operations
  - replacing system call table entries (knark)
  - replacing VFS fileops (adore-ng)

Example: system call table hijacking

```
orig_getuid = sys_call_table[__NR_getuid];
sys_call_table[__NR_getuid] = give_root;
```

Example: VFS hijacking

```
pde = proc_find_tcp();
o_get_info_tcp = pde->get_info;
pde->get_info = n_get_info_tcp;
```

Static Analysis of Kernel Module Binaries

- Symbolic execution
  - simulated program execution using symbols rather than actual input
  - machine state simulated as logical expressions using symbols
- Code sections of module disassembled and references to kernel symbols patched with actual values
- Initial machine state created, and symbolic execution begun from module initialization routine
  - machine state represented as set of registers, stack, and memory

Detecting Malicious Writes to Kernel Memory

- Kernel address loads *taint* destination register or memory
- Monitor writes to loaded kernel addresses or addresses *calculated from a loaded kernel address*
  - if destination address is not explicitly permitted by whitelist specification derived from legitimate kernel-module interface, write is labeled malicious

Example: detecting system call table hijacking

- kmodscan output

```
kmodscan: initializing scan for rootkits/all-root.o
[...]
kmodscan: DETECTED WRITE TO KERNEL MEMORY [c0347df0] at [.text+50]
[...]
kmodscan: 1 malicious write detected, denying module load
```

- offending instruction

```
50:  c7 05 60 00 00 00 00 00 00 00   movl   $0x0,0x60
```

- corresponding source line

```
sys_call_table[__NR_getuid] = give_root;
```

Example: detecting VFS hijacking

- kmodscan output

```
kmodscan: initializing scan for rootkits/adore-ng.o
[...]
kmodscan: DETECTED WRITE TO KERNEL MEMORY [c03e31b8] at [.text+d74]
[...]
kmodscan: 7 malicious writes detected, denying module load
```

- offending instruction

```
d74:  c7 40 20 00 00 00 00    movl   $0x0,0x20(%eax)
```

- corresponding source line

```
pde->get_info = n_get_info_tcp;
```

Challenges in Static Analysis Approach [1]

- Conditional branches
  - generally must explore both continuations of conditional branch
  - our system checkpoints machine state and executes one branch after another
  - results in exponential path explosion, mitigated by small size of module code
- Loops
  - without loop detection, symbolic execution would not terminate
  - however, cannot simply mark instructions as executed
  - we utilize dominator tree-based loop removal algorithm [Aho86]

Challenges in Static Analysis Approach [2]

- Data-dependent control flow
  - control flow targets may be based in part on program input and may be impossible to determine statically
  - possible to probabilistically determine targets, e.g. unreachable code analysis
  - our system currently labels module malicious and terminates execution, since no legitimate modules utilized unresolvable targets in experiments
- Approach does not consider /dev/kmem-based rootkits
  - userspace programs should not be allowed to write directly to kernelspace

Overview

- Motivation
- Kernel-Level Rootkit Detection
- **System Evaluation**
- Conclusions and Future Work

Experimental Setup

- Userspace prototype developed for Linux 2.6 kernels: kmodscan
  - analyzes ELF x86 modules
  - developed against two rootkits (knark, adore-ng)
- Detection capability evaluated against seven rootkits that implement a variety of different malicious functions
- False positive rate and performance overhead evaluated against entire Fedora Core 1 x86 default kernel module set

Detection Evaluation Results

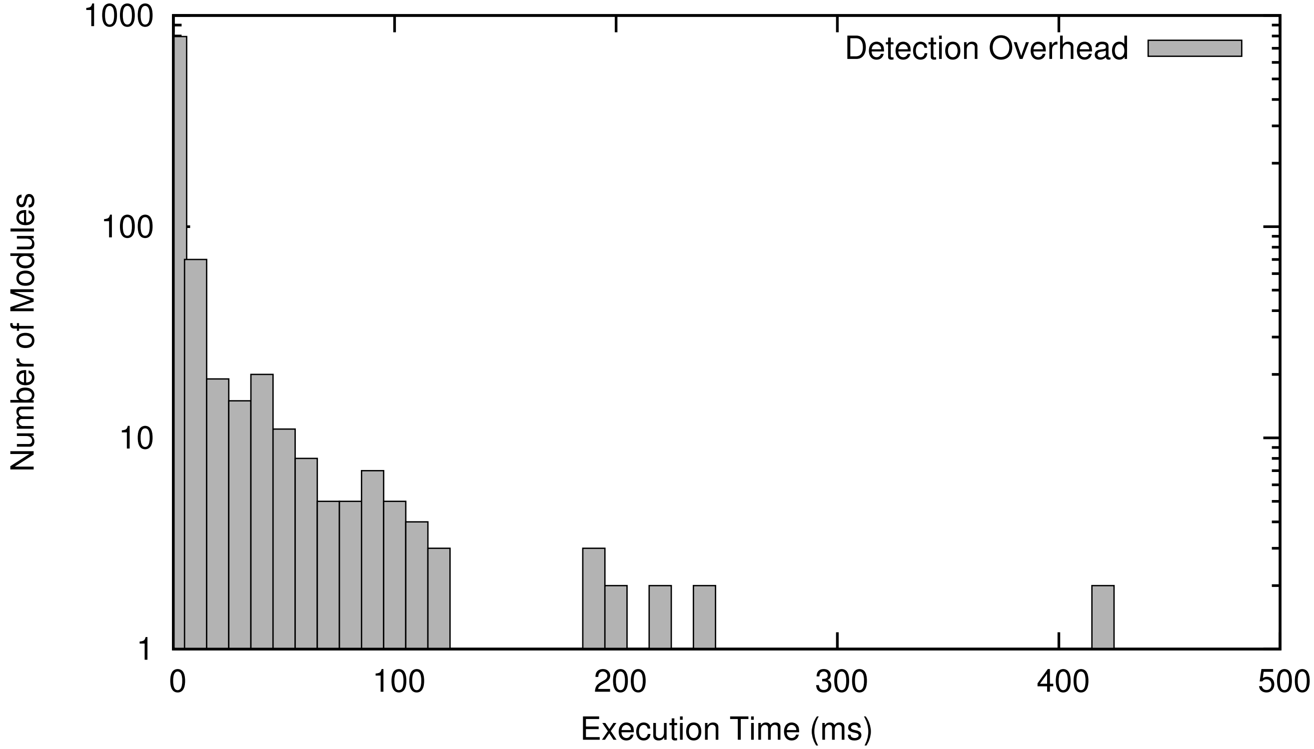
Rootkit	Technique	Description	Detected?
adore	syscalls	File, directory, process, and socket hiding Rootshell backdoor	Yes
all-root	syscalls	Gives all processes UID 0	Yes
kbdv4	syscalls	Gives special user UID 0	Yes
kkeylogger	syscalls	Logs keystrokes from local and network logins	Yes
rkit	syscalls	Gives special user UID 0	Yes
shtroj2	syscalls	Execute arbitrary programs as UID 0	Yes
synapsys	syscalls	File, directory, process, socket, and module hiding Gives special user UID 0	Yes

Module Set	Modules Analyzed	Detections	Detection Rate
Evaluation rootkits	7	7	100%

False Positive Evaluation Results

Module Set	Modules Analyzed	Detections	Misclassification Rate
Fedora Core 1 modules	985	0	0%

Performance Overhead Evaluation Results



Overview

- Motivation
- Kernel-Level Rootkit Detection
- System Evaluation
- **Conclusions and Future Work**

Conclusions

- Kernel-level rootkits are an increasing threat to system security
- Presented a behavioral specification-based kernel-level rootkit prevention mechanism enforced by binary static analysis
- Evaluted detection system against real-world Linux distribution
  - perfect detection rate against popular real-world kernel-level rootkits
  - low (non-existent) false positive rate against entire kernel module set for Fedora Core 1
  - low performance overhead

Future Work

- Formalize specification of kernel-module interface and behavior of kernel-level rootkits
- Increase sophistication of static analysis technique
- Integrate prototype into Linux 2.6 kernel module loader